CS 4973: DSA for Scalable Computing Spring 2025

Lecture 20 — March 26, 2025

Prof. Prashant Pandey *Scribe: Ben Lischin*

1 Overview

The previous lecture covered static graphs. We looked at graph representations including adjacency matrices, edge lists, adjacency lists, and compressed sparse rows. We introduced Ligra, a minimal graph API allowing for maximal optimization with concurrency in any graph algorithm.

In this lecture we shift towards streaming graphs and consider various ways to efficiently represent these dynamic structures.

2 Streaming Graphs

def: a **streaming graph system** receives a stream of queries/updates and must process both with low latency. The topology of a graph changes as we process inserts/deletes of nodes and edges.

More relevant terms for evaluating streaming graph performance:

- def: **throughput** - the number of operations completed in a unit of time. This can be improved with multi-threading
- def: **latency** - execution time of a single operation

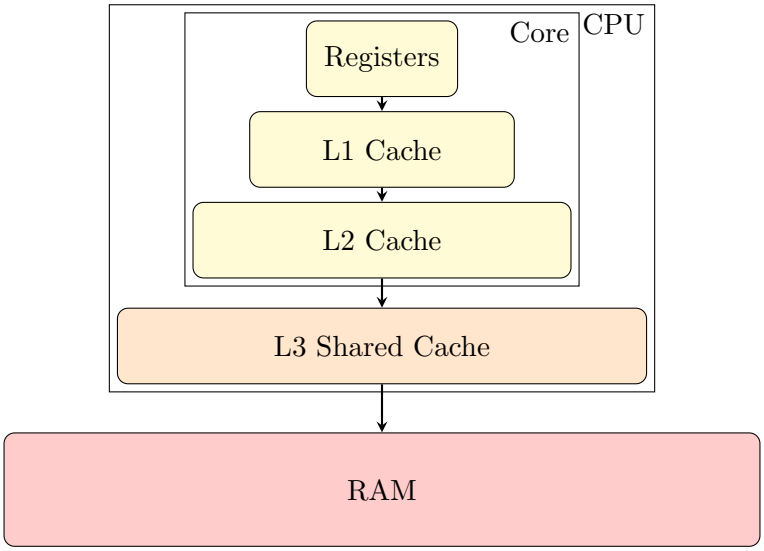
Existing streaming graph systems are generally categorized by one of the following two frameworks:

1. the system processes **phased** updates and queries in separate batches. Here, updates must be processed without query interference, so the system waits for queries before updating. These update batches block all threads to execute as fast as possible and then release the locks. This is the most common approach for streaming graphs we see in practice - graphs can be mutated without worrying about the underlying consistency of queries.
2. updates and queries are run **concurrently**, as enabled by *snapshotting*. To guarantee serializability, queries are isolated and run on a particular snapshot of the graph, while other updates run at the same time generate new snapshots. Consider that when topology is super critical to the algorithm, we want more frequent and fine-grained updates instead of batching (1).

2.1 Cache locality

Before we evaluate different streaming graph representations, we must first understand cache locality to discuss the tradeoffs of different approaches.

Consider the following low-level computer architecture for memory processing:



We will discuss two types of cache locality:

1. def: **spacial locality** has to do with organizing related data contiguously in memory to optimize the amount of relevant information retrieved by a single cache line request.

2. def: **temporal locality** relates more to algorithmic structure, optimizing our reuse of a given cache line before needing different data, encountering a cache miss, and replacing the cache line.

High-scale graph algorithm performance is dominated by the number of cache line retrievals. These I/Os are a large bottleneck due to the high-latency interaction between the cache and RAM. Whenever requested data is not in the cache, we encounter a cache miss which forces the CPU to stall while waiting for this data retrieval from RAM, reducing overall throughput. Optimized designs result in minimal cache misses and empirically faster execution in practice.

When data is stored in close proximity to other relevant pieces of information, more of them can be loaded into a single cache line from a particular RAM I/O due to cache locality. In other pointer-based approaches, we lose this advantage due to the sparsity of these items in memory, thus increasing the chance we will need to make another request to RAM.

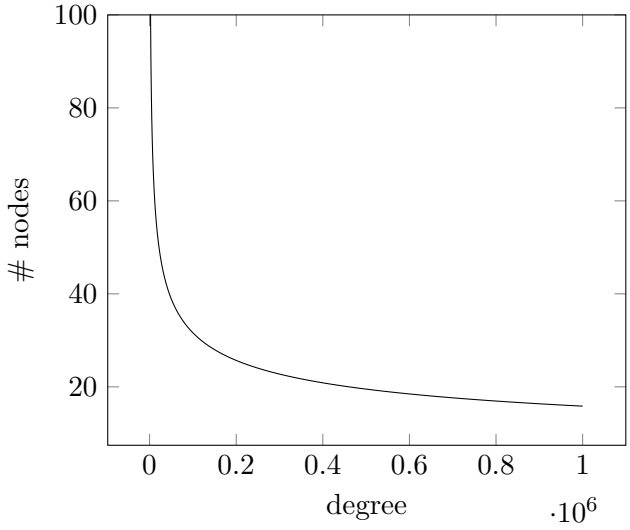
2.2 Structures for representing Streaming Graphs

1. **STINGER** [1] Ediger et al., HPEC 2012

   STINGER (Spatio-Temporal Interaction Networks and Graph Extensible Representation) is based on adjacency-list graph representation and supports both insertions and deletions of nodes and edges with fine-grained locking mechanisms. It is designed for dynamic graphs, allowing efficient updates while maintaining high concurrency by only locking smaller components of the graph.

Consider an implementation where we have 1M operations/sec. Scaling across x threads, we should expect a throughput of xM opeations/sec. This ideally what we want, but it does happen like this in practice. With STINGER's linked-list structure, performance can be bound by contention around a hotspot.

We know that real-world graphs are highly skewed and follow a power log distribution:



Nodes with many edges will hog the memory lock, becoming a limiting bottleneck. Listing the neighbors of these high-traffic nodes is very slow: we must follow a long linked-list of neighbors, following pointers to traverse each node. Even in the low-degree nodes with smaller neighbor lists, we will still likely encounter a cache miss for every single pointer link due to the nodes' scattered representation in memory.

As we will explore next, snapshots allow streaming graphs to scale better with threads.

2. **LLAMA** [2] Macko et al., ICDE 2015

LLAMA (Low-Latency Adaptive Multi-granular Architecture) is similar to STINGER, but specifically designed for batch update processing. With this, it is well-suited for single-writer, multi-reader patterns, where a single thread updates the graph while multiple readers can concurrently query it using all remaining threads without interrupting the update process. LLAMA notable supports snapshots to detangle the update and query processes. Each batch of updates creates a new snapshot, for incremental modifications that future queries may now use. This also makes LLAMA good for historical analysis of a graph. The storage overhead of these snapshots is $O(n)$ space to store the vertex array, where n is the number of nodes in the graph, and $O(k)$ space to store edge updates, where k is the number of edges modified by the batch.

3. **Aspen** [3] Dhulipala et al., PLDI 2019

Aspen is another snapshot-based system that supports batch processing, allowing queries to be executed consistently without being affected by any ongoing updates. It can be used as either a phased or fully dynamic system. Aspen uses a tree-of-trees model with *purely functional balanced search trees* to scale efficiently while maintaining faster search operations over large, dynamic graphs:

```
for each node in tree of nodes                          ▷ Search over the tree of nodes do
    for each neighbor in tree of neighbors              ▷ Search over the tree of neighbors do
        Process neighbor
    end for
end for
```

But standard balanced binary search trees are not optimal regarding spacial locality - nodes may be sparse in memory and we may face many cache misses loading each as we traverse pointers. To efficiently implement this tree-of-trees model, Aspen introduces the purely functional C-Tree.

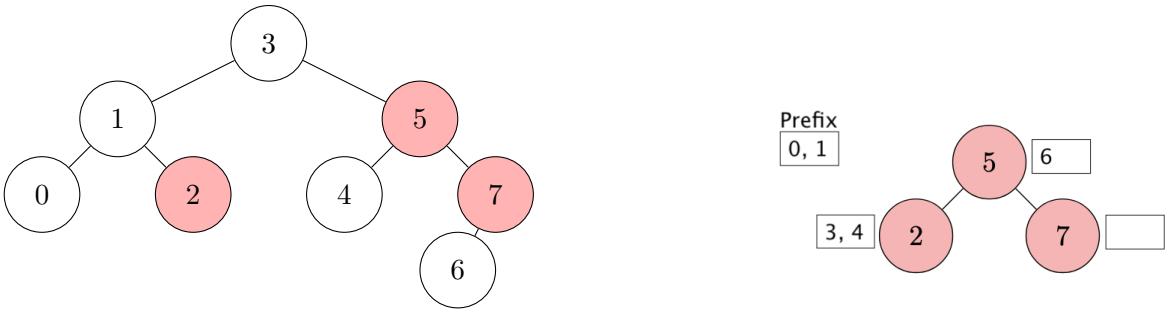
def: **purely functional trees** are immutable and rely on persistent data structures, where modifications create a copy without manipulating the original structure. In our context, we will never any in-place updates, any operation will use a pointer to a snapshot of a tree. Here, you pass snapshots by acquiring a pointer to the root of the node tree.

def: the **C-Tree** is a compressed purely functional tree that improves the locality of standard purely functional trees. To achieve this, the structure uses chunking and compression within each chunk. C-Trees are much better with concurrency across snapshots - we only need to do a root pointer swing after completing an update to ensure serializability and no race conditions. In practice, we only operate on minimal subtrees to avoid holding a lock on the entire tree when updating. For this, we copy the root-to-leaf path for where the update happens, to then mutate and swap back.

C-Tree construction:
We first start with a BBST and hash all items to randomly select tree heads for the C-Tree. For each head, we will store its tail: a contiguous array in memory of all nodes between this head and the next. Each of these tail chunks is further compressed in the tree to save space By the nature of our hash function, this guarantees the same heads will be picked in similar trees, ensuring consistent topology across all tree snapshots that we want to merge

```
h : K → {1, ..., N}                                     ▷ hash function signature
H(E) = {e ∈ E | h(e) mod b = 0}                         ▷ hash to generate heads
for each e ∈ E do
    tail(e) = {x ∈ E | e < x < next(H(E), e)}           ▷ find the tail of each head
end for
```

See the below visualization of the process detailed above. C-Tree heads are highlighted in red:



(a) Balanced Binary Search Tree → (b) C-Tree

4. **Terrace** [4] Pandey et al., SIGMOD 2021

   Terrace utilizes a hierarchichical storage of edges, exploiting the skewed power log degree-distribution of edges in real-world graphs, representing different degree classes in optimized ways:

   - low-degree nodes ($d < 15$) → neighbors in a Compressed Sparse Row
     - Store related nodes contiguously in place to load into a single cache line. Considering 64 Byte cache lines, we can store 16 x 4 Byte integers (the current node and 15 neighbors). With this setup, low-degree nodes will have no cache misses.
     - Note that by the power log distribution, the vast majority of our nodes will be represented super efficiently by this CSR structure.
   - medium-degree nodes ($15 < d < c \cdot 1000$) → neighbors in a **Packed Memory Array**
   - high-degree nodes ($d > c \cdot 1000$) → neighbors in a B$^+$-Tree
     - Efficient representation of large trees, especially on-disk. Items are all in linked leaf nodes, with internal nodes containing array partitions to enable faster traversal and efficient memory usage. This is the asymptotically optimal representation of high-degree nodes.
     - Note that by the power log distribution, relatively few nodes will be represented this way.

   Nodes can be moved between levels dynamically as the graph topology changes.

   def: **Packed Memory Array** (PMA) [5] Bender et al., 2006

   A Packed Memory Array is an array-based order maintenance data structure. Representing N items, we use...

   - $O(N)$ space
   - $O(\log^2 N)$ amortized updates
   - $O(\log N)$ queries

   The Packed Memory Array, an implicit complete BST on its cells grouped into leaves of size $\log N$, achieving good cache locality. The bottom structure of cells has length $c \cdot N$, c being a factor for padding space within the leaves. We utilize c extra space to buffer the leaves so we do not have to resize and shift the array on every insert which would be very costly (achieving the efficient amortized update time). This structure sees the benefits of contiguous array memory, but reduces the cost of inserts.

References

[1] David Ediger, Robert McColl, E. Jason Riedy, David A. Bader. STINGER: High performance data structure for streaming graphs. *HPEC*, 1–5, 2012.

[2] Peter Macko, Virendra J. Marathe, Daniel W. Margo, Margo I. Seltzer. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. *ICDE*, 363–374, 2015.

[3] Laxman Dhulipala, Guy E. Blelloch, Julian Shun. Low-Latency Graph Streaming using Compressed Purely-Functional Trees. *PLDI*, 918–934, 2019.

[4] Prashant Pandey, Brian Wheatman, Helen Xu, Aydin Buluc. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. *SIGMOD*, 1372–1385, 2021.

[5] Michael A. Bender, Haodong Hu. An adaptive packed-memory array. *PODS*, 20–29, 2006.